THE UNIX COMMAND LANGUAGE

K Thompson

Computing Science Research Dept
Bell Laboratories

K THOMPSON has a BS and an MS in Electrical Engineering from the University of California, Berkeley (1965 and 1966 respectively). He has been a member of the Technical Staff in the Computing Science Research Department of Bell Laboratories from 1966 to the present. His work has centred on languages, games, and systems. His early work has been on MULTICS, file system design and simulation, computer chess, cubic (3D) tic-tac-toe, and language implementation. His recent work has revolved around the UNIX time-sharing system and its subsystems software.

THE UNIX COMMAND LANGUAGE

INTRODUCTION

A program is generally exponentially complicated by the number of notions that it invents for itself. To reduce this complication to a minimum, you have to make the number of notions zero or one, which are two numbers that can be raised to any power without disturbing this concept. Since you cannot achieve much with zero notions, it is my belief that you should base systems on a single notion.

This paper describes the command language and some utilities on the UNIX time-sharing system (137), which runs on DEC PDP-11/45 computers. Examples are given that show how the single notion of I/O streaming and interconnection of utility programs are used to build complex programs from simple utilities.

Virtually every construction profession (building, hardware, etc) except that of software production relies on pre-fabricated building blocks. We have long since passed the stage when total rebuilding was feasible, yet this is still the way in which many systems are developed.

This paper, as well as describing the syntax and semantics of the UNIX command language, attempts to show a method of decomposition of programs and reconstruction using basic building blocks. This method is not perfect, or even right, but has to be a step in generating programs of the next level of complexity.

THE SHELL

Most users of UNIX, when they log on, find themselves talking to a program called the Shell. The Shell's job is to execute programs specified by the user. The simplest of the requests has the form:

$$command\ arg_1\ arg_2\ \ldots\ arg_n$$

This instructs the Shell to execute the named command and make the specified arguments available to that command as character strings. Since users may write their own commands and give them arbitrary names, the command name is first assumed to be a user-supplied name. In this way, the existence of an obscure system command does not restrict the user's vocabulary.

Every command executed has two files pre-opened for it. One can be read from (the standard input) and the other can be written to (the standard output). Normally, a program's standard input is the user's console keyboard, and the standard output is

the console printer. Thus, most commands use the standard I/O files for interactive communication with the user.

The Shell has the ability to redirect a command's standard I/O files without the command's knowledge. For example, the syntax:

```
sh <filename
```

instructs the Shell to run the sh command with the standard input switched to the file. This happens to be the name of the Shell itself. Thus this invokes a second Shell to read and execute the commands specified in the file.

The obvious extension of this syntax:

```
ls >output
```

instructs the Shell to run the ls command with standard output redirected to a file called **output**. This file will be created by the Shell if it does not already exist. If it exists, it will be truncated. The ls (abbreviation of list) command writes an alphabetical list of the names of all of the files in the current working directory. It is totally unaware of the redirection of the standard output that has occurred.

Another instance of output redirection

```
date >>output
```

instructs the Shell to put the standard output from the date command on the end of the **output** file. This is done by not truncating the **output** file if it is found to exist. If it does not exist, it is still created. In this case, the current date is appended to the file containing the directory list.

In UNIX, all I/O devices have names identical in syntax to regular files. This gives added power to the Shell's redirection abilities. For example,

```
sh <crd >lpr
```

starts a Shell reading from the card reader and writing to the line printer. This can naively be called 'batch processing'.

Commands, together with their I/O redirections, may appear on a single line if they are separated by a semi-colon (;) or an ampersand (&). If a command is followed by a semi-colon, its execution must complete before the rest of the line is interpreted. The previous examples of ls and date could have been executed by:

```
ls >output;  date >>output
```

With an ampersand instead of a semi-colon, there is a difference. The command preceding the ampersand is set in asynchronous execution. As in a previous example, the command

```
sh <crd >lpr&
```

will start a 'batch' job stream without tying up a console.

The most exotic feature of the Shell is its ability to connect the standard output of one command directly to the standard input of another. Again, neither program is aware that such things are going on. In the example

```
ls | pr
```

the command ls will produce a directory listing. The command pr will paginate this listing with dated headings. The pr command, of course, is just reading its standard input and is unaware that it is paginating anything but a stream of characters. This can be done less elegantly by the commands

```
ls >junk
pr <junk
```

followed by removal of the temporary file. Extending this idea, the command

```
ls | pr | lps
```

will cause a paginated directory listing to be delivered to the line printer spooler. The lps spooler copies its standard input to files that will eventually be copied to the line printer.

A final example of the Shell's syntax is its use of parentheses. Anything in matched parentheses is executed as a single command by a new instance of the Shell.

Thus,

```
(ls;  date) >output
```

will spawn another Shell to execute the commands inside the parentheses. This is identical to a previous example using >>. Another example is

```
(sleep 3600;  echo DONE)&
```

In this case, the sleep command will suspend execution for the number of seconds given in its argument. The echo command will write its literal argument onto the standard output. The combined effect is that after an hour the word DONE will be printed on the console. Of course, in the meantime, the console is free to execute other commands.

The Shell, and the commands it executes, form an expression language, the elements of which are quite grandiose operations. Since these operations can be user-supplied, it is easily extensible.

THE SHELL AS A COMMAND

The Shell is just another command and by redirecting its standard input, it is possible to execute commands from files. This section describes utility programs that are mainly used inside such command files.

First, and trivially,

```
: argument
```

does absolutely nothing. The command : is the much talked about 'null' program; it ignores its arguments and simply exits. One of its uses is to annotate Shell sequences.

The command

goto *argument*

'rewinds' its standard input. It then reads the standard input looking for the syntax of a : command with an argument matching its own. It leaves the standard input positioned after the : command. When control is returned to the Shell, execution continues where the standard input was positioned. (This implies certain things about file positioning and open file sharing that will not be discussed here.)

For example, if the text

```
: loop
command
sleep 120
goto loop
```

is placed in a file, then the command

```
sh <file
```

will execute whatever is specified by *command* about every two minutes.

The next example is

if *expression command*

The if command evaluates an expression consisting of one or more of its arguments. If the expression is true, it executes the rest of its arguments as a command. The expression may concern such things as the existence of files, comparisons of strings, and the successful execution of commands.

The Shell, as a command, may be passed arguments. If it is, these arguments may be substituted into the standard input like macro arguments. The Shell also allows for its whole list of arguments to be shifted. This makes it possible to repeat the same set of commands for each argument of the list.

The Shell also has the ability to construct argument lists from pattern matches on all the file names in a directory. In this way, it is possible, for example, to remove all files ending in 'x' or to print all files having three-character names, and similar operations.

All this gives the Shell and the commands that it executes the appearance of a programming language.

EXAMPLES

This section consists of further examples of Shell facilities followed by explanations of the commands used.

Spoken output

Consider the example

```
date | snobol spdate | speak
```

The **date** command produces output of the form:

```
Wed Jan  8 21:36:39 EST 1975
```

The second command runs the SNOBOL program **spdate**, which produces the following output:

```
today is wednesday january eighth.
at the tone the time will be
nine thirty six p m.
beeeeeeeeeep.
```

The last part of the example is the program **speak (138)**, which is a remarkable program that heuristically converts arbitrary English words into phoneme output for a commercially available voice synthesizer. So this example causes the computer to pronounce the time of day over a loudspeaker in the computer room.

Desk calculator operations

A more practical use of the **speak** program is

```
dc | numb | speak
```

dc is an arbitrary precision desk calculator. Since its output consists completely of numbers, the **numb** program converts digits into spoken numbers. The use of **speak** here is obvious; the effect is a desk calculator for the blind.

The input

```
2 32 ^
```

(reverse Polish meaning 2 to the 32nd power) causes dc to send

```
4294967296
```

to **numb**, which sends

```
four billion,
two hundred ninety four million,
nine hundred sixty seven thousand,
two hundred ninety six.
```

to **speak**, which pronounces it.

Encryption

In the example

```
crypt password <in | program | crypt password >out
```

an arbitrary program is flanked by an encryption/decryption program. The program receives clear input from the decryption of the input file. The program's clear output is then encrypted. This scheme allows the program to manipulate clear text maintained in encrypted form without the risk of first generating a clear file.

Selective directory listings

A selective listing of a user's files could be obtained by

```
(ls; tp t) | sort | uniq u
```

The **ls** program produces a list of the files in the current directory. The program **tp** maintains users' personal archive tapes. With the **t** option, **tp** produces a list of the files contained on a tape. The parentheses and semicolon concatenate the outputs of the two lists, which are sorted and delivered to **uniq**, which compares adjacent lines of text for equality. Conceptually, **uniq** splits a single copy of all duplicated lines onto a **d** stream and all other lines (unique lines) onto a **u** stream. The **u** option selects the **u** stream to be output. Thus the example will print all files that are found only on tape *or* only in the current directory. All file names found both on tape *and* in the directory are suppressed. The variants

```
((ls; tp t) | sort | uniq u); tp t) | sort | uniq d
((ls; tp t) | sort | uniq u); ls) | sort | uniq d
```

will further select the output. The first example will print file names that are on tape but not in the directory. The second will print those in the directory but not on tape.

Text formatting

Mathematical typesetting can be performed by

```
eqn <text | troff
```

The **troff** program accepts text and formatting directives for the purpose of photo-typesetting a document, and **eqn** (139) is a language pre-processor that translates equation descriptions into formatting instructions. Some examples of input and output are shown in Figure 1.

Equation description	*Output*
`{x sup 2} over {a sup 2} + {y sup 2} over {b sup 2} = 1`	$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$
`e sup {x sup {2 alpha} + y sup {2 beta}}`	$e^{x^{2\alpha} + y^{2\beta}}$

Figure 1: Mathematical typesetting

There are many more examples of language pre-processors. The desk calculator described above has an algebraic language front end. The FORTRAN compiler has a facade

implementing modern control structure (140).

Curve plotting

The sequence

```
calc | plot
```

is a simple example of a calculation that produces as output a set of paired numbers. The plot program accepts numbers, and produces a scaled labelled plot. If calc did not produce the numbers in monotonic order, the variant

```
calc | sort | plot
```

would fix that. Also if smoothing is necessary, the further amendment

```
calc | sort | spline | plot
```

includes a spline curve fitting algorithm.

Inter-user messages

Finally, the sequence

```
cc prog | mail cperson
```

shows the unexpected and extemporaneous way two programs were used. In this example cc is the compiler for the C language, which is the axis around which all of UNIX resolves. UNIX itself is written in C, as are the Shell, C, and about 70% of the system commands.

The mail command is used to send messages to a user's 'mailbox' file. The user is informed of the existence of this 'mailbox' when he first logs on.

What happened was that the person who maintains C accidentally installed a bad compiler late at night. When the morning crowd arrived, virtually all work was at a standstill. This example shows how one user sent twelve pages of undeserved C diagnostics to the culprit by way of a subtle complaint about the bug.

RETROSPECT

It is probably obvious that many of the examples of large computing tasks have the form:

source | *filter* | *sink*

Here the *source* program has something to say; ls and date are UNIX source programs. (Often the source is replaced by a file, which states it contents.) This is often processed by zero or more *filter* programs from where it is delivered to a *sink*. The sink is usually defaulted to the console, but can be an output file or a program resembling a pseudo-device. Sort and uniq are filters and speak and plot are pseudo-device sinks.

When new UNIX software is required, the prospective program can usually be recognized as belonging to one of these categories. As a result of this classification and a realization of a program's relationships to existing programs, much more general software is produced. Programs do not contain large number of features. These features are split out into separate filter programs. For example, our language processors do not produce listings or cross references; these are unrelated tasks.

The whole notion encourages a cleaner understanding of the anatomy of data processing. Programs are smaller, cleaner, easier to document, and easier to maintain.

SERMONETTE

Many familiar computing 'concepts' are missing from UNIX. Files have no records. There are no access methods. User programs contain no system buffers. There are no file types. These concepts fill a much-needed gap. I sincerely hope that when future systems are designed by manufacturers the value of some of these ingrained notions is re-examined. Like the politician and his 'common man', manufacturers have their 'average user'.